(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE INSTRUMENTATION OF A PARALLEL AND SCALABLE DATABASE COMPUTER--THE MULTI-BACKEND DATABASE COMPUTER, FOR BENCHMARKING ITS COMPLEX OPERATIONS

by

Darrell W. Alston

December 1989

Thesis Advisor: David K. Hsiao

 8

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION NAVAL POSTGRADUATE SCHOOL 6b OFFICE SYMBOL (If applicable) 37	7a NAME OF MONITORING ORGANIZATION NAVAL POSTGRADUATE SCHOOL
6c ADDRESS (City, State, and ZIP Code) MONTEREY, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) MONTEREY, CA 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification) THE INSTRUMENTATION OF A PARALLEL AND SCALABLE, DATABASE COMPUTER--THE MULTI-BACKEND COMPUTER, FOR BENCHMARKING ITS COMPLEX OPERATIONS

12 PERSONAL AUTHOR(S)
ALSTON, DARRELL W.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
MASTER'S THESIS	FROM _____ TO _____	1989, DECEMBER	82

16 SUPPLEMENTARY NOTATION THE VIEWS EXPRESSED IN THIS THESIS ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE DEPARTMENT OF DEFENSE OR THE U.S. GOVERNMENT.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
INSTRUMENTATION, PARALLED DATABASE COMPUTER, ~~MULTI-BACKEND DATABASE COMPUTER, BENCHMARKING~~, PERFORMANCE ~~EVALUATION~~, COMPLEX OPERATIONS

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

THIS STUDY IS THE CONTINUATION OF A PERFORMANCE EVALUATION TECHNIQUE KNOWN AS **BENCHMARKING** TO AN EXPERIMENTAL DATABASE MANAGEMENT SYSTEM KNOWN AS THE MULTI-BACKEND DATABASE SYSTEM (MBDS). THE MAIN EMPHASIS OF THIS THESIS IS ON THE **INSTRUMENTATION** OF THIS PARALLEL AND SCALABLE DATABASE COMPUTER, FOR BENCHMARKING ITS COMPLEX OPERATIONS: UPDATE AND RETRIEVE-COMMON. THE PRIMARY RESEARCH QUESTION IS TO DETERMINE WHETHER MBDS DEMONSTRATES THE RESPONSE-TIME REDUCTION AND RESPONSE-TIME INVARIANCE CLAIMS WHEN CARRYING OUT ITS TWO COMPLEX OPERATIONS. IN ORDER TO BENCHMARK THESE TRANSACTIONS, THE PROPER INSTRUMENTATION OF THE TEST DATABASE, TEST TRANSACTION SETS, AND TEST PROCEDURES WERE THOROUGHLY EXECUTED.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL PROF. DAVID K¢ HSIAO	22b TELEPHONE (Include Area Code) (408) 646-2253 22c OFFICE SYMBOL 52Hq

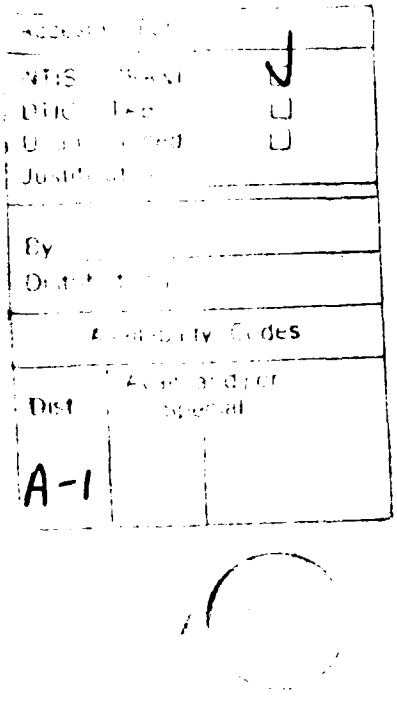
DD Form 1473, JUN 86 *Previous editions are obsolete* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

UNCLASSIFIED

LINE #19 (CONT.)

DETAILED TESTING, PROBLEM IDENTIFICATION (DEBUGGING), AND MINOR SOFTWARE MODIFICATIONS WERE ALSO CONDUCTED IN AN ATTEMPT TO VERIFY THE CORRECTNESS OF THE PROGRAM CODE FOR THE UPDATE AND RETRIEVE-COMMON OPERATIONS. MAJOR PROBLEM AREAS ARE DOCUMENTED, AND PROPOSED SOLUTIONS APE PRESENTED TO AID FUTURE EFFORTS IN THE EVALUATION, MODIFICATION, AND TESTING OF THESE EXTREMELY COMPLEX OPERATIONS TO ENSURE THEIR SUCCESSFUL PERFORMANCE EVALUATION.



Approved for public release; distribution is unlimited.

THE INSTRUMENTATION OF A PARALLEL AND SCALABLE DATABASE COMPUTER--THE MULTI-BACKEND DATABASE COMPUTER, FOR BENCHMARKING ITS COMPLEX OPERATIONS

by

Darrell W. Alston
Captain, United States Army
B.S., South Carolina State College, 1980

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

Author: Darrell W. Alston

Approved by: David K. Hsiao, Thesis Advisor

C. Thomas Wu, Second Reader

Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

This study is the continuation of a performance evaluation technique known as **benchmarking** to an experimental database management system known as the Multi-Backend Database System (MBDS).

The main emphasis of this thesis is on the **instrumentation** of this parallel and scalable database computer, for benchmarking its complex operations: UPDATE and RETRIEVE-COMMON. The primary research question is to determine whether MBDS demonstrates the response-time reduction and response-time invariance claims when carrying out its two complex operations. In order to benchmark these transactions, the proper instrumentation of the test database, test transaction sets, and test procedures were thoroughly executed. Detailed testing, problem identification (debugging), and minor software modifications were also conducted in an attempt to verify the correctness of the program code for the update and retrieve-common operations. Major problem areas are documented, and proposed solutions are presented to aid future efforts in the evaluation, modification, and testing of these extremely complex operations to ensure their successful performance evaluation.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

1. The Field of Study 1

a. Traditional Mainframe-Based Database Systems 2

b. Software Single-Backend Database Systems 3

c. Software Multiple-Backend Database Systems 4

2. The Previous Research 6

3. Our Area of Research 6

4. The Research Environment 6

5. The Importance of Research 7

B. COMPLEXITY OF A MULTIPLE-BACKEND DATABASE SYSTEM 8

C. RESEARCH QUESTIONS AND OBJECTIVES 10

1. MBDS Performance Claims 10

2. The Primary Research Question 10

D. SCOPE OF THE THESIS 11

E. ORGANIZATION OF THE THESIS 11

II. THE MULTIPLE-BACKEND DATABASE SYSTEM (MBDS) 13

A. MBDS DESIGN FEATURES 13

1. The Backend Controller 13

2. The Communications Bus 14

3. The Backend Computers 14
4. The Database Layout 15
a. Data Placements 15
b. Clustering 16
c. The Physical Distribution of Records ... 16
B. THE ATTRIBUTE-BASED DATA MODEL 17
1. The Base Data 18
2. The Meta Data 19
C. THE ATTRIBUTE-BASED DATA LANGUAGE 19
1. The Update Request 20
2. The Retrieve-Common Request 20
D. THE MBDS HARDWARE 21
1. Generic Unix Computers 21
2. The Disk Drives 21
3. The Broadcast Bus 21
III. THE INSTRUMENTATION PROCESS USING "CABS" 23
A. AN OVERVIEW OF CABS 23
B. SETTING UP THE TEST DATABASE 24
1. The Test Database Generator 24
a. Configuring the Test Database 24
b. The Record File 26
c. The Template File 28
d. The Descriptor File 29
2. Limitations on the Test Database Size 29
C. SETTING UP THE TEST TRANSACTION MIX 29
D. HOW TO USE CABS 31

1. Running the Program 31

2. The Output 33

a. The Database Files 33

b. The Transaction Mix Files 34

c. The Report Files 35

E. THE BENCHMARKING METHODOLOGY 35

1. Initializing the System Setup 36

2. Operating the Test Interface 36

a. Starting the System 36

b. Loading the Test Database 37

c. Conducting Performance Testing 39

d. Collecting the Performance Data 42

e. Exiting the Test Interface 43

IV. INSTRUMENTATION OF THE COMPLEX OPERATIONS 44

A. THE RETRIEVE-COMMON OPERATION 44

1. An Overview of Retrieve-Common 44

a. An Operation on Two Files 44

b. The Syntax of Retrieve-Common Operation 45

2. The Design Approach 46

3. The Implementation Based on Bucket-Hashing 46

4. The Logical Operation 47

5. The MBDS Test Transactions 48

6. Existing Problems with the Retrieve-Common Operation 49

a. The Hashing Procedure 51

b. Problems with the Bucket Number Constant 52

c. Other Problems Encountered with Retrieve-Common 53

7. Proposed Solutions to Existing Problems 55

B. THE UPDATE OPERATION 56

1. An Overview of Update 56

2. The Design and Execution of an Update Request 56

3. The MBDS Test Transactions 59

4. Existing Problems with the Update Request .. 59

5. Proposed Solutions to the Problems 61

V. THE PRELIMINARY RESULTS 62

A. SUMMARY OF RESULTS 62

1. The Single-Backend Configuration 63

2. The Two-Backend Configuration 64

3. The Three-Backend Configuration 64

4. The Four-Backend Configuration 65

B. A COMPARATIVE CHART 66

VI. CONCLUSIONS AND RECOMMENDATIONS 68

LIST OF REFERENCES 71

INITIAL DISTRIBUTION LIST 73

I. INTRODUCTION

A. BACKGROUND

1. The Field of Study

With the influx of modern computer technology, data processing has become a highly significant operation of many government, private, and commercial organizations. Most organizations rely on timely, accurate information to aid in control, management, and decision-making. The need for fast, accurate, efficient and economical information processing has motivated an unending interest in **database systems** research.

Database systems are special-purpose computers which consist of both hardware components and specialized software packages called **database management systems (DBMS)**. Consequently, database systems are usually referred to as database management systems. These systems own and control their own databases, disk subsystems (secondary storage), and transaction libraries, allowing them to perform on-line, data-intensive transactions associated with real-world database management applications.

The traditional data processing using, files, tapes, and manual handling of transactions by operators is being replaced by modern database management using databases, disks, and automatic handling of transactions by database management systems [Ref. 1].

As a result of influential database systems research, three database-system approaches have emerged. These include the traditional mainframe-based approach, the software single-backend approach, and the software multiple-backend approach [Ref. 2].

a. Traditional Mainframe-Based Database Systems

In a **traditional mainframe-based** database system, the DBMS software runs on a large mainframe computer, sharing the computer's resources (CPU, secondary storage, etc.) with other executing application programs. (See Figure 1.) As a result, the performance of a mainframe-based database system decreases as the mainframe workload increases.

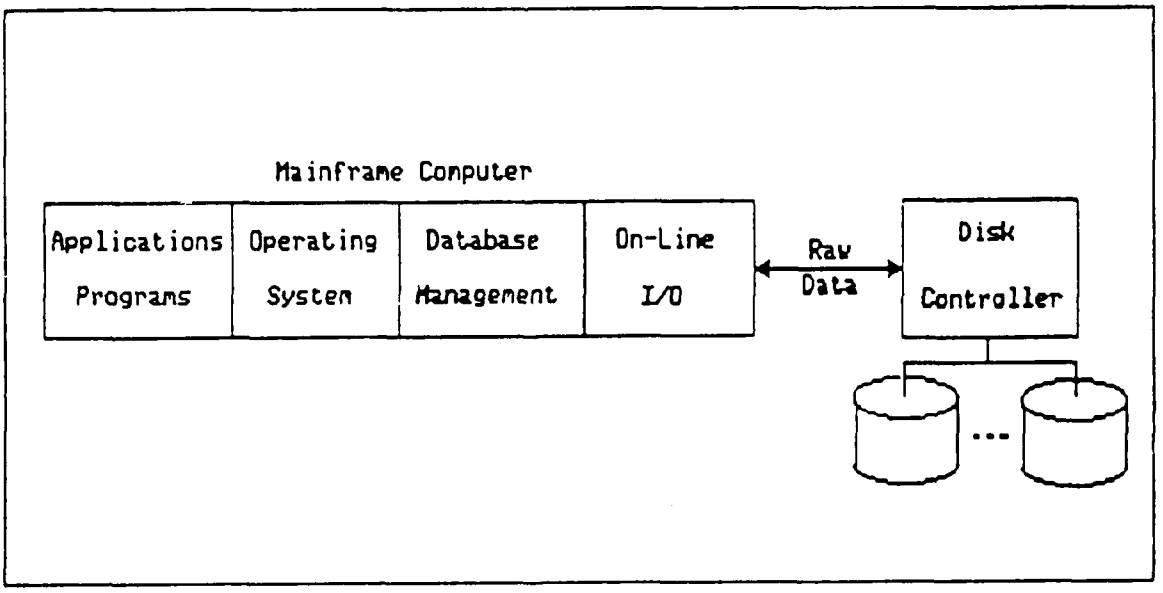


Figure 1. Traditional Mainframe Approach [Ref. 3]

b. Software Single-Backend Database Systems

The **software single-backend** database system evolved as a conventional approach to solving the problems of performance degradation and resource sharing encountered by the mainframe-based system. In the single-backend approach, DBMS runs on a separate, dedicated computer with its own operating system and disk system (see Figure 2). This database computer known as a **backend** processor, is connected to one or more host (frontend) mainframe computers via a two-way communications link. By moving the database management function to the backend computer, we free up the busy frontend mainframe computer to process other application

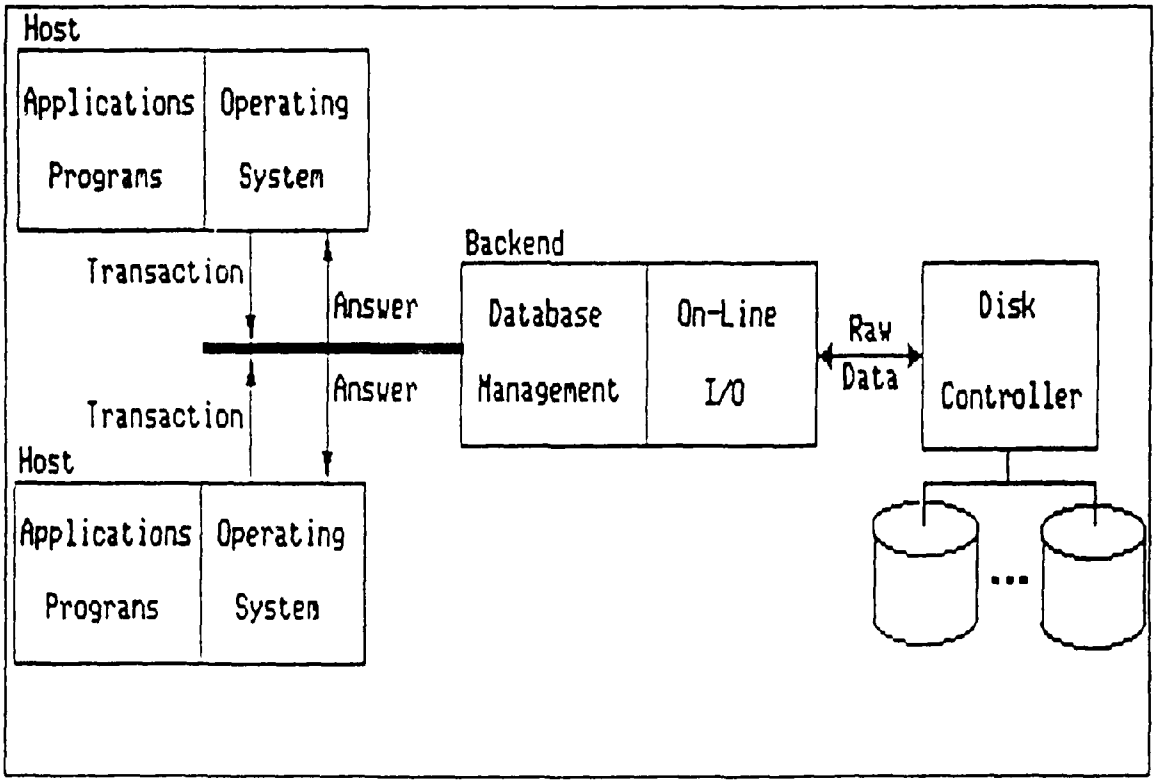


Figure 2. Single-Backend Approach [Ref. 3]

programs. The frontend computer receives user requests in the form of database transactions and transmits them to the dedicated backend database computer. These transactions are processed by the database computer and the response is returned to the user over the communications link. The single-backend database system is more efficient and cost-effective than the traditional mainframe-based database system. However, the performance of the single-backend database system remains measurably degraded as the workload on the backend computer increases.

c. Software Multiple-Backend Database Systems

To overcome the performance and upgrade problems experienced with both the traditional and the single-backend systems, an unconventional approach, known as the **software multiple-backend** database system has been developed. In this approach the database system consists of at least one **backend controller** and two or more **backends** interconnected by a communications bus. The controller controls transaction processing of the backends, and interfaces with the host mainframe computer. The backends with their own disk systems and identical software, perform the requested database operations on the database which is distributed across the disk systems. (See Figure 3.)

Multiple-backend database systems are capable of providing both high-performance database management and large-capacity growth. Increasing the number of backends and

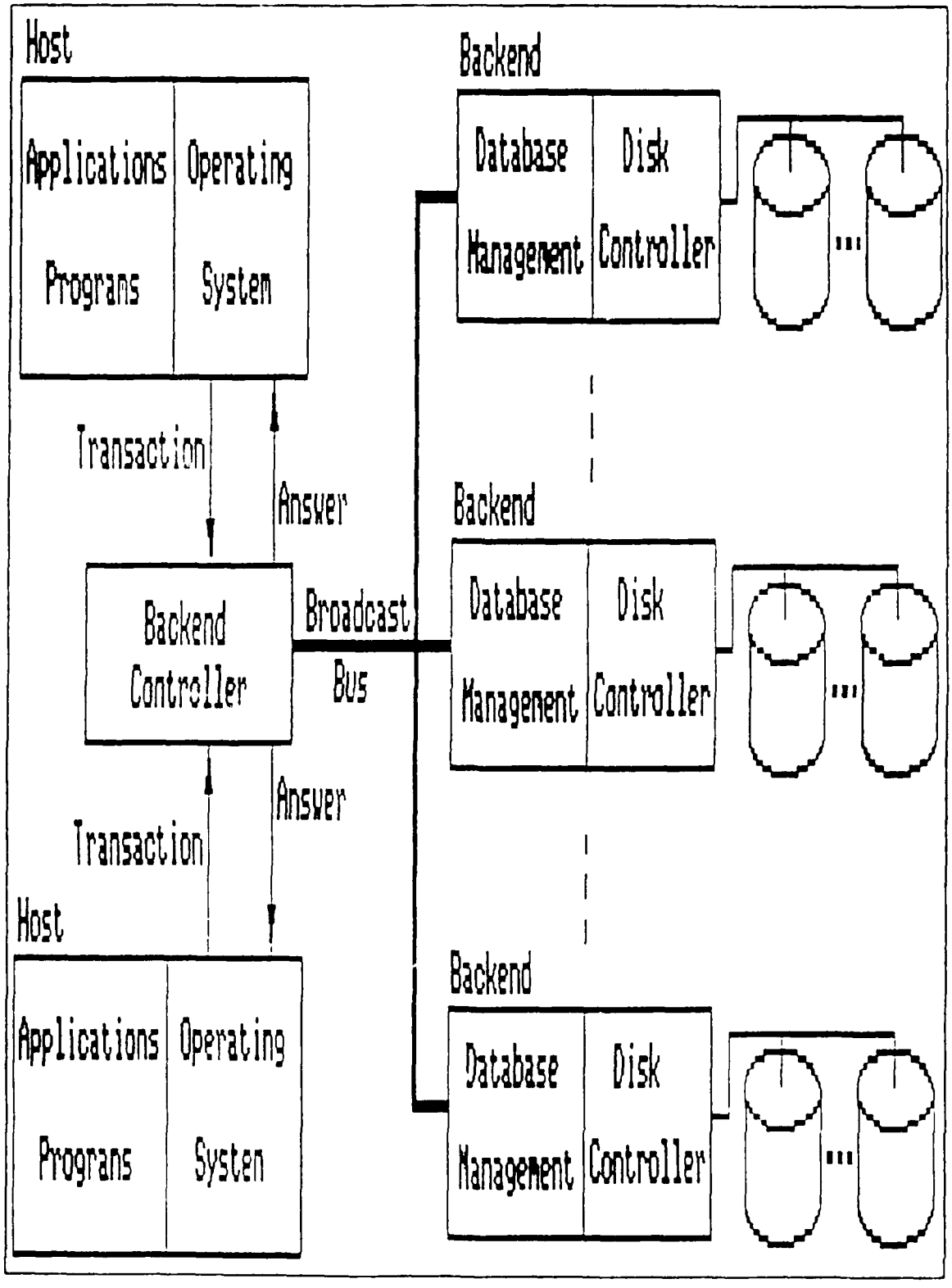


Figure 3. Multi-Backend Approach [Ref. 3]

distributing the database evenly over each of the backend's disk subsystems should demonstrate predictable performance gains and capacity growth in a multiple-backend system [Ref. 3].

In order to determine the performance capabilities of a given multiple-backend configuration, a formal method for measuring system performance must be exercised. This study is a continuation of the performance evaluation presented in a recent thesis by James Hall [Ref. 3] on an experimental multiple-backend database system which is under development in the Laboratory for Database Systems Research of the Naval Postgraduate School.

2. The Previous Research

Prior to James Hall's thesis, various significant research efforts were responsible for the development of the methodology and computer-aided design (CAD) tools used in this study. Hall's thesis presents a chronological list of previous work, briefly describing the contribution of each.

3. Our Area of Research

Our area of research involves the performance evaluation of parallel database computers. This study is the actual application of a performance evaluation technique called **benchmarking** to an experimental database management system known as the Multi-Backend Database System (MBDS) in an attempt to verify the implementor's claims of MBDS in terms of performance gains and capacity growth. However, in this

thesis we focus mainly on the instrumentation of this parallel database computer for benchmarking its complex operations: **Update and Retrieve-Common.**

4. The Research Environment

Research was conducted in the NPS Laboratory for Database Systems Research on a network of modern UNIX (ISI) workstations. Hall used the same controlled environment to conduct his research. The research was aided by using the same existing set of computer-aided design (CAD) tools for performance evaluation known as the Computer-Aided Benchmarking System (CABS) [Ref. 4]. Maintenance and development of this experimental Multi-Backend Database System (MBDS) is still being conducted by professional programmers.

5. The Importance of Research

James Hall in his recent thesis completed the benchmarking (testing) of MBDS with eight (8) parallel backends. His benchmarking was on two of the four set-oriented database operations, namely, the **RETRIEVE and DELETE** operations, which are the two most simple operations. Using the benchmarking results collected, Hall was extremely successful in verifying the performance claims of the designer and implementor of MBDS in terms of response-time reduction (performance *gains*), *response-time invariance* (capacity growths) measures.

There are two other set-oriented database operations: **UPDATE and RETRIEVE-COMMON.** These two database operations are

far more complex than the retrieve and delete operations. Thus, prior to any benchmarking, MBDS must be prepared for such undertaking. Such preparation is termed **instrumentation**. Although this research is to follow the tradition established by Hall's thesis, its main focus is on the instrumentation of MBDS for applying the benchmarking methodology onto these two extremely complex operations. Subsequently, the results of the benchmarks can be used to **fully** assess the performance claims of the implementor of MBDS based on all four of its operations: RETRIEVE, DELETE, UPDATE, and RETRIEVE-COMMON.

B. COMPLEXITY OF A MULTIPLE-BACKEND DATABASE SYSTEM

There are three design requirements of MBDS that makes it far more complex than either the conventional mainframe-based system or the more recent single-backend system. These design requirements are discussed in [Ref. 5] as follows:

The first requirement states that a multi-backend database system must be expandable in order to support the addition of backends for performance enhancements and capacity growth. This expansion must require no modification to the existing database software, no new programming necessary for the expansion, no modifications to the hardware and no major disruption of system activity when additional backends are being incorporated into the system. The system is also **scalable** in that a variable number of backends must be tested for valid, effective performance evaluation.

The second requirement mandates that both the hardware and software are **generic**. The hardware of the backends should be typical and readily available and can be added to the system with minimal interruption of the system activity. The backend software should be designed so that a new backend can be integrated into the system by simply replicating the database system software of another backend into the new backend. With this requirement, a multi-backend database system can be upgraded by adding new backends of the same type and by using existing system software.

The third requirement suggests that for storage a database is **evenly distributed** across the disk systems of the backends, and for operation there are **parallel and concurrent processing** of transactions by the backends. Thus, when a transaction is being processed, a backend works on its own portion of the database in parallel with other backends working on their own portions of the same database. This is parallel processing of a transaction and parallel access to the database. In addition to parallel processing and access, the backends process several transactions concurrently in order to overcome any idling of backends and delay in accessing the database. By exploiting the parallelism and concurrency of the backends and by distributing a database evenly for storage, the system should gain in performance due to parallel accesses.

C. RESEARCH QUESTIONS AND OBJECTIVES

1. MBDS Performance Claims

MBDS is designed to employ two or more backends for parallel processing of transactions, by distributing the database evenly over the backends' disks. Performance gains (in terms of response-time reduction) and capacity growth (in terms of response-time invariance) of a multi-backend database system are likely to be proportional to the number of backends of the system. The following is a summary of the performance claims that have been made by the designer and implementor of MBDS [Ref. 3]:

* **Response-Time Reduction (RTR).** The response-time reduction of a transaction is inversely proportional to the multiplicity of the backends. This means that as the number of backends increases, the response-time reduction for a given transaction is expected to improve (e.g., moving from one backend to three backends should yield a response-time which is just one-third the original response time).

* **Response-Time Invariance (RTI).** The response-time invariance of a transaction in response to the increase of the database size is maintained by a corresponding increase in the multiplicity of the backends (e.g., when the database doubles in size, doubling the number of backends will yield the original response-time).

2. The Primary Research Question

The primary research question is to determine whether MBDS demonstrates the response-time reduction and response-time invariance when carrying out its two complex operations: UPDATE and RETRIEVE-COMMON. In order to benchmark these transactions we must ensure the proper instrumentation of the test database, test transaction sets, and test procedures.

We must also verify the correctness of the program code for these transactions through detailed testing, problem identification (debugging), and software modification.

D. SCOPE OF THE THESIS

The scope covers the broad spectrum of performance evaluation and design analysis of database computers in parallel architecture. However, the main emphasis will be on the instrumentation of a parallel database computer, namely, MBDS, for benchmarking its complex operations.

A benchmarking methodology will be applied to MBDS with a variable number of parallel database processors (backends). The benchmarking will be used to identify any design flaws and implementation drawbacks encountered while processing the complex transactions. Proposed solutions to identified problems will be presented in order to correct or modify system software for efficient performance evaluation. Some preliminary benchmarking results have also been provided where the initial instrumentation process has been successful.

E. ORGANIZATION OF THE THESIS

The remainder of this thesis is organized into five chapters. Chapter II is an overview of MBDS. We describe the design features of MBDS, the attribute-based data model, the attribute-based data language, and the MBDS hardware. In Chapter III we elaborate on the Computer-Aided Benchmarking System (CABS) and its use in the instrumentation process.

Chapter IV contains a detailed description of the two complex operations: UPDATE and RETRIEVE-COMMON. We describe the algorithmic approach of each operation, existing problems found within the algorithms, and proposed solutions to the problems to ensure proper instrumentation of the system. In Chapter V preliminary test results are provided to verify the instrumentation.

Lastly, in Chapter VI we present a summary of the thesis, present conclusions, and offer suggestions for future work on the benchmarking of MBDS.

II. THE MULTIPLE-BACKEND DATABASE SYSTEM (MBDS)

A. MBDS DESIGN FEATURES

1. The Backend Controller

Although MBDS can be configured using a single backend, the most effective performance results are achieved by configurations consisting of two or more backend computers with one computer acting as the backend controller.

As required by the software multiple-backend approach, the MBDS backend controller does very little work. It is mainly responsible for:

* receiving and pre-processing user transactions,
* transmitting the transactions to all of the backends simultaneously for execution,
* collecting and post-processing the transaction results,
* routing the results to the host or terminal, and
* arbitrating data insertion into the database by the backend.

During pre-processing, user transactions are reformatted and placed on the broadcast bus. The post-processing function combines the records received from the backends in response to a transaction and performs any aggregate operations (AVG, SUM, etc.) requested before forwarding the complete results to the user [Ref. 3].

2. The Communications Bus

MBDS uses a **broadcast bus** to perform its communications tasks. A broadcast bus was selected over other bus topologies primarily because it allows easy expansion of the number of parallel backends on the system. The task of adding another backend to a given system simply requires the connection of the backend computer's communications to the local-area network which is connecting the MBDS machines [Ref. 3].

By using a broadcast bus, MBDS can achieve parallel execution of user transactions. The controller transmits a message to all of the backends via the broadcast bus and the message is received almost simultaneously by the backends. The broadcast bus is also used by a backend to communicate with all the other backends when necessary and, primarily to return the partial result of transactions to the controller for post-processing.

3. The Backend Computers

The backends with their own disk systems and software are the workhorses of MBDS, performing the requested database operations on the database which is distributed across the disk systems. Each backend operates only on its portion of the database and returns a partial result to the controller for post processing. The following description of the design of MBDS backends is extracted from [Ref. 5]:

> ...the backends of the system all have identical software to allow replication of the software on a new backend.

Additionally, the backends must have complete software to perform all of the database management functions. These functions include directory management, concurrency control, record processing, and communications. The directory management function is responsible for managing indices, calculating record clusters, allocating the secondary-storage addresses for record insertion, maintaining secondary-storage tables of indices, cluster numbers, and addresses, processing transactions against the directory tables, and providing record addresses for subsequent database access operations. The concurrency control function oversees various accesses to the directory tables and the user data and facilitates the concurrent execution of transactions. The record processing function is used to stage the user data from the secondary storage to the primary memory, to process the staged data, to store data onto the secondary storage, and to return the responses to the controller. Finally, there are communication functions in each backend to control communications among backends and between the backend and the controller. It is necessary to minimize the communications among backends, in order to reduce the communications traffic among them.

4. The Database Layout

a. Data Placements

The full performance potential of MBDS depends on the even distribution of the database across the backends. The design of the MBDS database is extracted from [Ref. 5]:

> In a multi-backend database system, a database must be placed on the secondary storage in such a way so that all of the subsequent accesses to the database will result in block-parallel-and-record-serial operation. In other words, all of the backends are accessing, in parallel, the secondary-storage blocks of the same database in their respective disk systems, although the records in the blocks which may satisfy the same transaction or different transactions are being accessed by the backends serially. Thus, the issue really focuses on how to ensure an even distribution of the user database across the disk systems of the backends. Such a distribution requires a data placement algorithm. To achieve an even distribution of data, there must be a processor in the multi-backend database system that is responsible for overseeing the record-insertion process. The controller has an overview of the entire system, and is the logical choice for

arbitrating the record insertion process, i.e., controlling the data placement.

b. Clustering

To achieve block-parallel-and-record-serial operation, MBDS partitions logically related records into **clusters of records** [Ref. 3]. The clusters consist of one or more blocks of storage space. A **block** is a preset unit (e.g., a track) of the backend's secondary storage space.

The cluster-based database placement is arbitrated by the controller and carried out by the backends [Ref. 5]. New clusters are formed by the backends. When the first record is to be inserted to form a new cluster, the controller uses its data placement algorithm to randomly select the starting backend for the cluster. The backend in turn, allocates a block of the secondary storage (disk) space to store the record. Under the direction of the controller, the chosen backend will continue to add new records of the same cluster into the block until the block of storage space is filled. The backend then notifies the controller that the block is full. In a round-robin manner, the controller directs another backend to continue the placement of new records of the same cluster.

c. The Physical Distribution of Records

During Hall's research it was discovered that the uneven distribution of records across the backends is

inevitable. The following is the physical distribution of records as described in Hall's thesis [Ref. 3]:

> ...this clustering methodology can easily cause a certain amount of uneven loading. This is especially true when the block size is much larger than the record size or the number of records in a given cluster is small. This uneven loading phenomena is virtually unavoidable because of the random selection of the first backend of each cluster and the variable number of records possible in each cluster.
>
> Because the selection of the first backend of a given cluster is a random decision, it is possible that certain backends could be selected more often than others....
>
> MBDS uses pseudo-random number generator from the UNIX system function library and like any good pseudo-random number generator it tends to pick, on the average, a number near the middle of the range involved.... This has a noticeable effect on the distribution of records.

Although a small amount of uneven loading of records was encountered, it had little impact on the overall performance of MBDS.

B. THE ATTRIBUTE-BASED DATA MODEL

The **attribute-based data model** is the database model used by MBDS. It provides a high-level abstraction which allows the user to focus on the logical properties of the database without being concerned about the implementation and instrumentation details of the database and the database system.

In the attribute-based data model, the data is considered in the following constructs: database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record

body, keyword predicate, and query [Ref. 5]. These constructs are applied to two kinds of data: the **base data** and the **meta data.**

1. The Base Data

The following description of the base data is extracted from [Ref. 5].

> Informally, a database consists of a collection of files. Each file contains a group of records which are characterized by a unique set of directory keywords. A record is composed of two parts. The first part is a collection of attribute-value pairs or keywords. An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. As an example, <POPULATION,25000> is an attribute-value pair having 25000 as the value for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Certain attribute-value pairs of a record (or a file) are called the directory keywords of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in a directory for identifying the records (files). Those attribute-value pairs which are not kept in a directory are called non-directory keywords. The rest of the record is textual information, which is referred to as the record body.

In MBDS, the database consists of several record files. The records are stored in secondary storage as a collection of attribute-value pairs followed, optionally, by a record body. However, the record body feature has not been implemented. An example of a record is shown below.

(<FILE, USCensus>,<CITY,San Jose>,<POPULATION,40000>, {Mild Climate})

The angle brackets <,> enclose an attribute-value pair. The curly brackets {,}, enclose the record body. The first attribute-value pair of all records of a file is the same.

The name of the attribute is **FILE** and the value is the file name (NOTE: in this study the attribute named **FILE** is actually implemented as **TEMP**). This first attribute-value pair is mandatory as it partitions the database by the file name.

2. The Meta Data

The **meta data** is the stored information about the base data. We depend on the meta data for quick, direct access of desired records for a transaction. MBDS meta data is made up of attributes, descriptors, and clusters, and are known as the directory information. Attributes represent types of base data. Descriptors are used to specify, either ranges of values (Type-B) or exact values (Type-A) of the attributes. A cluster is a group of records such that every record in the cluster satisfies the same set of descriptors.

The directory information is organized in three tables: the **attribute table (AT)**, the **descriptor-to-descriptor-id-table (DDIT)** and the ***cluster-definition table*** **(CDT)**. AT maps directory attributes to the descriptors defined on them in the descriptor-to-descriptor-id table. DDIT maps each descriptor to a unique descriptor id. CDT maps descriptor-id sets and record ids to cluster ids.

C. THE ATTRIBUTE-BASED DATA LANGUAGE

The attribute-based data language (ABDS) is the basis of the data language of MBDS which supports the five primary

database operations: INSERT, DELETE, UPDATE, RETRIEVE, AND RETRIEVE-COMMON. In this thesis, we focus only on the two complex operations. UPDATE and RETRIEVE-COMMON.

1. The Update Request

An UPDATE request is used to modify records of the database. The UPDATE request consists of two parts, a query part and a modifier part. The query identifies the records of the database to be updated (changed) and the modifier specifies how the records are to be updated. The following UPDATE request will modify all records of the student file having course number C200 by changing the GPA to four (4).

```
UPDATE((TEMP=Stud)and(CNUM=C200))<GPA=4>
```

2. The Retrieve-Common Request

The RETRIEVE-COMMON request is used to merge two files by common attribute values. The format of the RETRIEVE-COMMON request is given below:

```
RETRIEVE (query-1) (target-list-1)
COMMON (attribute-1, attribute-2)
RETRIEVE (query-2) (target-list-2)
```

Attribute-1 and Attribute-2 in the COMMON clause are the attribute names associated with the first RETRIEVE request and the second RETRIEVE request, respectively. The records in each file that satisfy both the query and the COMMON clause are selected. The following is an example of a RETRIEVE-COMMON REQUEST.

```
RETRIEVE (TEMP=Cors)(PLAC,ROOM)
COMMON (CNUM,CNUM)
RETRIEVE (TEMP=Stud)(NAME,GPA)
```

This request would find all records in the Cors (course) file and all records in the Stud (student) file that have the same (common) attribute value, CNUM (course number), and return the PLAC (place) and ROOM, student NAME and GPA of the respective records having the same CNUM.

D. THE MBDS HARDWARE

1. Generic Unix Computers

The computers used during this study as well as Hall's study are the Integrated Solutions, Inc. (ISI) super-micro-computers. There are nine (9) ISI computers each using the 4.3 BSD UNIX operating system. Each computer also uses a 16.67 MHz Motorola CPU and has four megabytes of main memory. They are compatible to SUN 3 workstations.

2. The Disk Drives

All of the backend computers have three hard-disk drives: two small disk drives and a big disk drive. One small disk is for the operating system to do paging and filing. The other small disk with its 100 megabytes of storage space is used to store the MBDS meta data. The big disk has a storage capacity of 400 megabytes and is used to store the MBDS base data. These disk drives are connected to the backend's internal, high-speed data bus.

3. The Broadcast Bus

All of the backends and the controller are connected to a local-area network (LAN). MBDS uses the industry-standard Ethernet communications bus as its LAN, although

additional MBDS software has been incorporated into the LAN for reliable broadcasting. The controller serves as a gateway to other external computers at the Naval Postgraduate School.

III. THE INSTRUMENTATION PROCESS USING "CABS"

A. AN OVERVIEW OF CABS

The Computer-Aided Benchmarking System (CABS) is a computer-aided design (CAD) tool, developed as a result of several highly significant research efforts. Previous work by Strawser [Ref. 6], Tekampe and Watson [Ref. 7], and Vincent [Ref. 2] led to the actual implementation of CABS by Fenton [Ref. 4].

CABS is extremely critical to the performance evaluation of MBDS because it provides an automatic means of generating the test database sets, and the test transaction mixes for benchmarking parallel, multiple-backend database systems. Without CABS the time and effort required to produce these key features would be immeasurable.

The MBDS user needs only to input three essential elements of information to CABS in order to generate a test database and a test transaction mix. These elements of information include:

* the number of backends in the system,
* the disk track size in the system, and
* the maximum disk storage of a single backend.

With the maximum number of backends as input, the test database is designed to be distributed evenly over each of the possible MBDS test (backend) configurations. Special

descriptor files are created by CABS to accomplish this distribution, along with the raw data records to insert in order to build the database [Ref. 3]. The test transaction sets generated by CABS are based on the database size submitted as input by the user.

B. SETTING UP THE TEST DATABASE

1. The Test Database Generator

The objective of the database generator is to pass to a TEST directory named "UserFiles," all of the necessary files that represent the three database sizes recommended by [Ref. 2]: Large (N bytes), Medium (N/2 bytes), and Small (N/4 bytes). The directory contains three types of database files, the template file, the descriptor file, and the record file.

CABS uses Strawser's recommendation to use four different record sizes: Large (N), Medium-Lg (N/2), Medium (N/4), Small (N/10) [Ref. 3]. Each of the smaller record sizes evenly divides the large record size. All three of the test databases are made up of twenty-five percent (25%) of each of the respective record sizes.

a. Configuring the Test Database

The following information representing the configuration of test databases is extracted from [Ref. 3].

The first step in finding a database of size N, which can be distributed evenly between a given number of backends, is determining the least common multiple (LCM) of

the number of backends to be tested. CABS uses a lookup table of precalculated LCMs. Using the LCM in the database calculation ensures that for each configuration, the number of backends can be divided evenly into the number of records used.

CABS uses the LCM to calculate the size in bytes of the smallest database building block which can be split evenly between the backends. This building block is known as the database multiple (DBM). The DBM is a multiple of 32 [Ref. 2], because the database must be divisible by four since the database has to be quartered into the four different record sizes. The small database size is one quarter the original (N) size database. Finally, the implementation of CABS was simplified by using database size divisible by two. The calculation to arrive at 32 as the factor was simply (4 * 4 * 2) = 32, which ensures the divisibility of the DBM. The last element of the DBM calculation is the large record size which is completely system dependent. In Strawser's original scheme [Ref. 6], the large record size is based on the disk track size. Unfortunately, modern disk drive capacity (30 or more kilobytes/track) makes such a record size impractical and unrealistic, as well, since database records are normally much smaller than 30,000 bytes. For this study a large record size of 1,000 bytes was chosen because it was large enough to force MBDS to fragment the records on the communication bus, but small enough to permit the use of a

scaled-down database. The actual database multiple for this study then became:

* DBM = LCM(1,2,3,4,5,6,7,8) * 32 * 1000 bytes
* DBM = 840 * 32 * 1000 bytes
* DBM = 26,880,000 bytes.

This was, in fact, as close as CABS could get to the scaled-down database target size of 30 megabytes. The three database sizes provided by CABS were:

* Large 26,880,000 bytes
* Medium 13,440,000 bytes
* Small 6,720,000 bytes.

Note that these calculations are actually carried out in base ten by CABS, apparently to simplify the actual calculations.

b. The Record File

The current version of CABS produces only one class of record files. This class of record files represents the response-time reduction (RTR) test. Each database (small, medium, and large) has its own unique RTR record file. According to [Ref. 4], CABS was supposed to generate a second class of record files which represents the response-time invariance (RTI) testing, but this feature was not implemented in the actual software that resulted from [Ref. 4]. RTR testing requires only one input file to be used with all of the RTR configurations of a given database because the size of database does not change. RTI testing requires an extra input file for each of the backends in the test (i.e., an

eight backend test requires eight input record files for the RTI testing) [Ref. 3].

The record file generated by CABS containing the input code and raw attribute-value data necessary to build the database for all four record sizes. The record file is the input for the Test Interface (TI) mass load utility [Ref. 3]. The first three fields of each record are the template (file) attribute and the two descriptor attributes. These attributes are directory keywords used to cluster the records. The first attribute, the template attribute, is a Type-B attribute which values are limited to a specific set of values. The template attribute which is referred to as "TEMP" is also the mandatory FILE attribute for each record. "TEMP" receives as a value one of four template names describing the record size class: large records **(Templg)**, medium-large records **(Tempmedlg)**, medium records **(Tempmed)**, or small records **(Tempsmall)**.

The second and third attributes, the descriptor attributes, are Type-A attribute which take on a range of possible values for each attribute. The names for the descriptor attributes consist of three parts. The first part is called INT which identifies the attribute value as an integer. CABS simply inserts the consecutive record number starting from one for each record class [Ref. 3]. The second part is called either ONE or TWO. ONE associates the descriptor attribute to the nine cluster categories. TWO associates the descriptor attribute to the number of records

per cluster. The third part identifies which record class, large (LG), medium-large (MEDLG), medium (MED), or small (SMALL). Thus, the second attribute (e.g., INTONELG) would be used to partition the database consisting of large records into nine cluster categories. The third attribute (e.g., INTTWOLG) would be used to partition the large record database into hundreds of smaller clusters. A fourth attribute called the MULTIPLE attribute is the first non-directory keyword attribute of a record. The values of this attribute are character strings representing how many times the database has been multiplied (e.g., "one", "two", "three", ..., etc.) [Ref. 4]. These values are used to distinguish between the RTI files. Since the RTI files are not generated, this attribute is not implemented in the current version of CABS, and was not used during this study.

c. The Template File

While the record file is fairly generic and could be used as the input file for other DBMS tests, the template and descriptor files are specifically created to support the loading of the various MBDS test configurations [Ref. 3]. CABS generates a common single template file which is shared by each of the database sets (small, medium, and large). This file contains four independent templates, with each template associated with a different record class. Each template contains the names of both the directory and non-directory

attributes and their associated data types, i.e., either string (s), integer (i), or floating number (f).

d. The Descriptor File

Each of the three database sets has its own unique descriptor file which contains indexing information for each directory attribute in the database. The file includes the name of each directory, its attribute value type (Type-A or Type-B), and its datatype. Type-B descriptors are followed by a list of the allowable values. Type-A descriptors are followed by a list of the attribute-value ranges that specify each cluster category. Taken together as a set of values for any given record the descriptors identify a unique cluster of records [Ref. 3].

2. Limitations on the Test Database Size

Due to the technical limitations discussed in Hall's thesis [Ref. 3], the previous performance evaluation of MBDS was conducted only on the smallest of the three database sizes provided for testing by CABS. The size of this small database is approximately seven (6.56) megabytes of base data consisting of 30,240 records. The small database (6.56 MB) used by Hall is the only database used in this study. The generation of a larger database is still not feasible.

C. SETTING UP THE TEST TRANSACTION MIX

The test transaction mix was designed to allow the performance evaluator to use one set of test transactions for

all of the test configurations of each of the three database sizes [Ref. 3].

So for each of the three database sets (large, medium, small) generated by CABS, a set of test transactions are generated. Each set of transactions has four subsets of transactions one for each record class (large, medium-large, medium, and small). Each subset of transactions consists of 24 different ABDL transactions which access a proportional number of records within their respective record class. The transactions are constructed from each of the five database operations, and categorized as either over-head intensive or data-intensive.

For this study CABS has generated three RETRIEVE-COMMON transactions for each of three record classes (large, medium-large, medium) in the small database set. Each of the three transactions accesses a different proportion of records in their respective record classes. These transactions can be found in the "UserFiles" directory of the backend controller under the following file names: SDB_LGR#1, SDB_MLR#1, and SDB_MDR#1.

However, it was discovered in this study that CABS did not generate any UPDATE transactions in the transaction sets. Therefore, the UPDATE transactions must be manually constructed and inserted into the existing transaction set or placed in a new file.

D. HOW TO USE CABS

1. Running the Program

To execute the program, the evaluator types the command "cad" at the UNIX system prompt in the "BENCH" subdirectory of the MBDS system. The user will then see the following prompt:

Input the number of backends in the system > 8

The evaluator must input the maximum number of backends to be included in the test. For this study the maximum number of backends is eight. This value should always be the total number of backends provided by the system.

The system will now respond with the track size prompt:

Input the disk track size in the system in bytes > 2000

The number (2000) entered at the prompt above indicates the size of the records used by CABS for the test database. This value is double the number of bytes the evaluator has set as the large record size (1000). It is also the block size assumed by CABS. [Ref. 3]

At the next prompt, the evaluator must input the maximum amount of data to be loaded to any single backend computer. The system prompt is:

Input the max disk storage of a single backend in whole megabytes (MBYTES) > 30

The evaluator must ensure there is adequate space in the UNIX file system for the record file size specified.

At this point, the generation of the record files, and the generation of reports are optional to the user. The following sequence of prompts and system responses permits evaluation of the reports before the lengthy record file generation is activated:

Do you want to generate the reports > y

Reports will be generated

Do you want the record files generated? (y/n) y

PERFORMING INITIAL CALCULATIONS

GENERATING THE TEMPLATE FILES

GENERATING THE DESCRIPTOR FILES

GENERATING THE RECORD FILE

Create (s)mall, (m)edium, or (l)arge record file? s

Creating a small record file

GENERATING THE TRANSACTION MIX FILES

GENERATING THE REPORT FILES

The above procedures in the CABS program are to be used for future requirements in generating a new test database

of a larger size. These procedures were not exercised during this study.

2. The Output

The following information describing the CABS output is extracted from [Ref. 3]:

CABS produces a total of 49 different files during execution and deposits the files in the directory from which the program is executed. There are three main groups of files produced:

* Database input files
* Transaction mix files
* Report files.

The first two groups of files in the directory can be listed with the standard UNIX "ls" command. The report files are hidden but can be listed by using the optional UNIX "ls -a" command. A list of these files appear in [Ref. 3].

a. The Database Files

The CABS files necessary to build the test databases are the following:

* TEST.t - the MBDS template file
* TEST.dl - the MBDS descriptor file (large database)
* TEST.dm - the MBDS descriptor file (medium database)
* TEST.ds - the MBDS descriptor file (small database)
* TEST.r - the base data to be inserted.

In order to load the test database, the evaluator has to manually copy or move the above files to the appropriate

directories. The "UserFiles" directory on the backend controller needs to contain the following files:

* TEST.t
* TEST.d
* TEST.r.

Note that there is only one descriptor file listed above. The evaluator must select the appropriate descriptor file for the database (small, medium or large) and rename it to TEST.d. The UserFiles directory on each of the backends must contain the following files:

* TEST.t
* TEST.d.

These files are the same as the ones on the controller.

b. The Transaction Mix Files

The ABDL test-transaction files created by CABS are:

* LDB_LGR#1, LDB_MLR#1, LDB_MDR#1, LDB_SMR#1
* MDB_LGR#1, MDB_MLR#1, MDB_MDR#1, MDB_SMR#1
* SDB_LGR#1, SDB_MLR#1, SDB_MDR#1, SDB_SMR#1.

Each file contains 24 transactions which is more than enough for a complete test. These files are text files which can be modified in a text editor, if a smaller set of transactions can meet the evaluator's needs. The file names can be changed by the evaluator, as well, but the naming convention using a pound sign followed by a number must be maintained. The transaction files are the input for the test interface (TI)

and are imported by using the TI "select transaction unit" option. These files must also be moved to the controllers UserFiles subdirectory.

c. The Report Files

All the remaining files in the directory are report files. The evaluator report files can be printed out by using the Unix command:

tbl <filename> ¦ psroff -me.

There are a large number of reports, but because they do not reflect the actual distribution of records, the reports are not very useful. The CABS files listed below do provide a good description of the "ideal" database topology which can be used as a comparison with the actual database:

* .ev_test_config_ldb
* .ev_test_config_mdb
* .ev_test_config_sdb.

E. THE BENCHMARKING METHODOLOGY

The Benchmarking Methodology of this study was used throughout the performance evaluation conducted by Hall, and was originally documented in his thesis [Ref. 3]. This methodology, as it is presented in the following sections, includes step-by-step procedures and user-interface (pop-up) menus which smoothly guides the evaluator through the benchmarking process.

1. Initializing the System Setup

Prior to loading a test database, the evaluator must ensure that the secondary storage devices are clear of any other data. The backend controller has a subdirectory named "test." This subdirectory is further subdivided into directories for each possible MBDS configuration and are appropriately titled 1, 2, 3, 4, 5, 6, 7, 8. By selecting the directory appropriate to the configuration under study, and executing the script-file named "zero" the evaluator may begin the benchmarking process. The command "zero" will cause the meta data and base data disks used by MBDS to be cleared. Another command called "zip" is provided in the above directories, and may also be used to carry out the same function as the "zero" command. Either process takes about one hour per backend.

2. Operating the Test Interface

a. Starting the System

To run the test interface (TI), the evaluator uses the UNIX "cd" command to move to the test subdirectory appropriate to the configuration under study. The evaluator then issues the "run" command located in the subdirectory. The "run" command will start TI, the MBDS processes on the controller and the processes on the appropriate backends. The evaluator must ensure that all six processes are activated on the controller, and all six processes are activated on each backend before the operation is continued. The initial

start-up takes some time because communications between the controller and backends must be established. Once MBDS is online and the TI main menu is presented, the evaluator selects the menu choice "a" as follows:

```
-------------------------------------------------------------

The Multi-Lingual/Multi-Backend Database System

     Select an operation:

          (a) - Execute the attribute-based/ABDL interface
          (r) - Execute the relational/SQL interface
          (h) - Execute the hierarchical/DL/I interface
          (n) - Execute the network/CODASYL interface
          (f) - Execute the functional/DAPLEX interface
          (x) - Exit to the operating system

     Select-> a

-------------------------------------------------------------
```

b. Loading the Test Database

To load the test database, the evaluator needs to select the Load-a-database option from the menu below:

```
-------------------------------------------------------------

     The attribute-based/ABDL interface:

          (g)  -  Generate a database
          (l)  -  Load a database
          (r)  -  Request interface
          (x)  -  Exit to the previous menu

     Select-> l

-------------------------------------------------------------
```

The next step is to load the test database template file by selecting the use-a-database option from the menu below and responding to the prompt for the database name with "test" which is the test database name as below:

```
Select an operation:

    (u)  -  Use a database
    (r)  -  Mass load a file of records
    (x)  -  Exit, return to previous menu

Select-> u

Enter the name of the database: test
```

At this point, the system is ready to begin the mass-loading of the record file. To accomplish this, the evaluator selects the mass-load-a-file-of-records option from the menu below and respond to the prompt with the record file name "TEST.r".

```
Select an operation:

    (u)  -  Use a database
    (r)  -  Mass load a file of records
    (x)  -  Exit, return to previous menu

Select-> r

Enter the record file name: TEST.r
```

The loading process will start and provide feedback on the progress of the mass-loading utility every ten records. Initially, the records load at a rate of about one megabyte an hour and after six to seven hours the UNIX operating system slows the entire process considerably. Since MBDS must run for many hours to build the databaase, it is

advisable to enter the following key strokes once the loading process is running to cause MBDS to end the session normally and save the meta data:

```
u     <CR>
test  <CR>
x     <CR>
x     <CR>
x     <CR>.
```

The operating system will buffer this input and accept the commands when the loading process finishes. This frees the evaluator from monitoring the progress of the mass-load process and limits the damage should one of the MBDS computers crash after a successful load. By ending normally, the system will write the meta data to secondary storage which permits restarting the system at a later date. An abnormal ending (such as a crash) after a successful mass-load could be costly because reloading the database is the only way to build the meta data again.

c. Conducting Performance Testing

Once the test database is loaded, the evaluator can begin the actual performance evaluation. The first step is to ensure that there are no other users on the controller nor on any of the backends. Next, the evaluator must start TI from the same system subdirectory used to load the test configuration. From the main menu below, the evaluator selects the execute-the-attribute-based/ABDL-interface option:

```
-------------------------------------------------------------

     Multi-Lingual/Multi-Backend Database System

     Select an operation:

          (a)  -  Execute the attribute-based/ABDL interface
          (r)  -  Execute the relational/SQL interface
          (h)  -  Execute the hierarchical/DL/I interface
          (n)  -  Execute the network/CODASYL interface
          (f)  -  Execute the functional/DAPLEX interface
          (x)  -  Exit to the operating system

     Select-> a

-------------------------------------------------------------
```

Next, selects the request-interface option from the menu below and respond to the prompt for the database name with "test":

```
-------------------------------------------------------------

     The attribute-based/ABDL interface:

          (g)  -  Generate a database
          (l)  -  Load a database
          (r)  -  Request interface
          (x)  -  Exit to the previous menu

     Select-> r
     Enter the database id: test

-------------------------------------------------------------
```

Next, selects the performance-testing option from the menu below:

```
-------------------------------------------------------------

     Select a subsession:

          (s)  SELECT:  select traffic units from an existing
                        list (or give new traffic units) for
                        execution
          (n)  NEW LIST:  create a new list of traffic units
          (d)  NEW DATABASE:  choose a new database
          (p)  * PERFORMANCE TESTING
          (r)  * REDIRECT OUTPUT:  select output for answers
```

(m) * MODIFY: modify an existing list of traffic units
(o) * OLD LIST: execute all the traffic units in an existing list
(x) EXIT: return to previous menu

Refer to the MLDS/MBDS user manual before choosing subsessions marked with an asterisk (*)

Select-> p

The next step is to enable the system timers; i.e., selects the turn-on-external-timer option from the menu below and then selects the exit-to-previous menu option as shown below:

Select an operation:

(e) Turn on external timer.
(i) Turn on internal timers.
(a) ABORT..Abandon all requested actions.
(x) Exit to previous menu.

Select-> e

External Timer On

Select an operation:

(e) Turn on external timer.
(i) Turn on internal timer.
(a) ABORT..Abandon all requested actions.

(x) Exit to previous menu.

Select-> x

The next step is to load a set of test transactions to run against the database. To accomplish this, the evaluator selects the first menu choice from the menu

below and respond to the prompt with the name of one of the four transaction sets appropriate for the size of the database under study:

```
-------------------------------------------------------------

     Select a subsession:

          (s)  SELECT:  select traffic units from an existing
                        list (or give new traffic units) for
                        execution
          (n)  NEW LIST:  create a new list of traffic units
          (d)  NEW DATABASE:  choose a new database
          (p)  * PERFORMANCE TESTING
          (r)  * REDIRECT OUTPUT:  select output for answers
          (m)  * MODIFY:  modify an existing list of traffic
                          units
          (o)  * OLD LIST:  execute all the traffic units in
                            an existing list
          (x)  EXIT:  return to previous menu

          Refer to the MLDS/MBDS user manual before choosing
          subsessions marked with an asterisk (*)

     Select-> s

          Enter the name for the traffic unit file
          It may be up to 40 characters long including the
          .ext.  Filenames may include only one '#' character
          as the first character before the version number.

          FILE NAME-> SDB_MDR#1

-------------------------------------------------------------
```

d. Collecting the Performance Data

Once the transaction set is loaded, the evaluator just needs to enter the transaction number (they are numbered from zero) at the menu below:

```
-------------------------------------------------------------

     Select Options:

          (d)     redisplay the traffic units in the list
          (n)     enter a new traffic unit to be executed
          (num)   execute the traffic unit at [num]
                  from the above list
          (x)     exit from this SELECT subsession

     Option-> 0

          (<CNT(INTONEMED, 160>)

     Start:  12:32:31     Stop:  12:32:33     Elapsed Time:
     2.167
-------------------------------------------------------------
```

The response time of a transaction is displayed as shown above. The evaluator must manually transcribe the times for later analysis. To move to the next record size and test transaction set, exits from this submenu, chooses the select option again and changes to the file name of the next test transaction set.

e. Exiting the Test Interface

To exit TI, uses the menu selections to end the test normally. When the user ends normally, all of the MBDS controller and backend processes are also stopped. Although exiting TI by using the CTRL-C keypress is possible, this leaves most of the MBDS processes running. They will interfere with those processes which are started by the next run of TI. If it is necessary to end abnormally, there is a script file in each configuration subdirectory named "burn" which will stop all MDBS processes.

IV. INSTRUMENTATION OF THE COMPLEX OPERATIONS

A. THE RETRIEVE-COMMON OPERATION

1. An Overview of Retrieve-Common

a. An Operation on Two Files

The Retrieve-Common request is used to merge two files by common attribute values. The **common-attribute values** are the attribute values of the records belonging to both files. The retrieve-common operation is defined and described in [Ref. 8] as follows:

Logically, the retrieve-common request involves two retrieval operations. The first retrieval operation is defined as the **source retrieve** and the second retrieval operation is defined as the **target retrieve**. The set of all the records that belong to the result of the source retrieve is called the **source record set**. The set of all the records that belong to the result of the target retrieve is called the **target record set**. A **source (or target) record** is the record that belongs to the **source (or target) record set**. Correspondingly, these attributes will be referred to as source (target) attributes. The merged source and target records are called the **result record set**.

The source and target attribute names that participate in the retrieve-common operations are referred to as **join attributes**, and their values **common attribute values**.

The retrieve-common operation requires that the join attribute which is specified in the source record set must have the same domain as that of the join attribute in the target record set, although they need not have the same attribute name.

b. The Syntax of Retrieve-Common Operation

The syntax for the retrieve-common request resembles the syntax of the ABDL retrieve request. This allows the actual selection of records from secondary storage to proceed as two retrieve requests [Ref. 9]. As depicted in Chapter II, the syntax for the retrieve-common is:

```
RETRIEVE (query-1) (target-list-1)
COMMON (attribute-1, attribute-2)
RETRIEVE (query-2) (target-list-2)
```

The retrieve-common request consists of three parts. The first part is referred to as the **source retrieve**, which retrieves the source record set. The second part consists of the join attributes, where Attribute-1 refers to the source record and Attribute-2 refers to the target record. The values of these two attributes must be identical in order to satisfy the condition for merging a source record with a target record. Although their values must be identical, their attribute names need not be identical. The third part is called the target retrieve, which retrieves the target record set. An example of a Retrieve-Common request is given below:

```
RETRIEVE (TEMP=US) (STATE, TOWN)
COMMON (TOWN, CITY)
RETRIEVE (TEMP=CANADA) (PROV, CITY)
```

This retrieve-common request would return, from the United States and Canada files, all states, towns, provinces, and cities, where the town and the city have identical names.

2. The Design Approach

The main issue when considering a design strategy for implementing the retrieve-common request is where the merge of the source and the target records should be performed. The design approach used for the MBDS retrieve-common request is to have the backend do the merge operation. Each backend performs a merge of its portion of the source records with all of the target records, including those target records sent to it by the other backends. Each backend then, sends its merged results to the controller which, in turn, forwards the final result to the user [Ref. 8].

This approach minimizes the controller function and allows even distribution of the workload to each backend.

3. The Implementation Based on Bucket-Hashing

The implementation discussed in this section is described in [Ref. 8]. This implementation strategy attempts to speed up the comparison and merge by hashing records into small groups (the buckets of the hashing tables) which contain records with common attribute values, so that the time complexity of the merging operation may be reduced.

A hashing function applied to the common attribute value is used to hash records into buckets. The bucket numbers are consecutive integers. Instead of using primary

and overflow areas, the buckets use one or more fixed-size blocks to store records. The numbers of blocks may vary among buckets.

Two separate hashing tables are used, one for the source record set, and one for the target record set. This alternative is accomplished in three phases:

* The backends will hash and store their own source records and target records into separate hashing tables by a common hashing function. After all of the target records have been hashed and stored, each backend will broadcast the hashed results of their target records (i.e., the bucket number and the records associated with that bucket number) to all of the other backends.

* Upon receiving all of the target information from the other backends, each backend stores those target records into appropriate buckets according to their bucket numbers.

* The backends perform the merge operation on the local source records and the local (i.e., the entire) set of target records and send the results to the controller.

4. The Logical Operation

After reviewing the design approach and the implementation strategy, the logical operation of the retrieve-common is listed as follows from [Ref. 9]:

* The retrieve-common request is converted into two retrieve requests by placing the common attributes into the target list of the source retrieve and the target retrieve, respectively.

* All of the records which satisfy the source retrieve are gathered, the common attribute value is hashed, the records placed in the virtual memory, and the hashed addresses are stored in the hash tables.

* All of the records which satisfy the target retrieve are collected, and the hash values are calculated. These records are also placed in the virtual memory and their addresses are stored into another hash table.

* The target records of a backend are transmitted to all of the other backends to be added to the local target records. In this way, each backend has only a portion of source records of the database, but has every target record which is in the database.

* To perform the pairwise merge, the backend checks if the value of the join attribute in each source record is the same as the value of the join attribute in each target record, since the join attributes have been specified in the request. If the two values are the same, the records are concatenated and outputted.

5. The MBDS Test Transactions

For this study CABS has generated three retrieve-common transactions within the test transaction files of each of three record classes (large, medium-large, and medium). Records from the small record class are retrieved from the target retrieve portion of each retrieve-common transaction in the test transaction files of the medium record class. Thus, retrieve-common transactions are not included in the test transaction files of the small record class. The test transaction files namely SDB_LGR#1, SDB_MLR#1, and SDB_MDR#1 are located in the "UserFiles" subdirectory on the backend controller (presently db8).

The retrieve-common test transactions perform merge operations on two back-to-back record classes, i.e., on large and medium-large, on medium-large and medium, and on medium and small. Each of the three transactions access a different proportion of records in their respective record classes. The first test transaction accesses a very small selection (two percent) of records in the first cluster category of

back-to-back record classes and return those records that share common INTONExx attribute values within each respective record class. This type of transaction is classified as overhead-intensive which means most of the time is spent looking up the clusters and on communications between the controller and the backends rather than reading the database. The second test transaction accesses all the records from the record class of both the source and target retrieves such that only one-half (1/2) the records satisfy each retrieve. Those records that have common INTONExx attribute values are retrieved and outputted. This type of transaction is called data-intensive because most of the time is spent accessing the data, not on processing them. The third test transaction accesses all the records from the record class of the source retrieve and that only one-half (1/2) the records satisfy that retrieve. For the target retrieve only one sixteenth (1/16) the records of that record class are accessed. Those records that have common INTONExx attribute values are outputted. This type of request is considered to be both overhead-intensive and data-intensive.

6. Existing Problems with the Retrieve-Common Operation

During the instrumentation of the Retrieve-Common operation, several major problems were discovered in the program code. The first major problem was discovered while trying to test the second and third test transactions of each of the three record files.

When using a multiple-backend configuration consisting of two backends or more, these transactions will not execute. When the transaction number is entered at the system prompt the system will attempt to process the transaction. The record process (recproc.tr) on the backends starts to execute (indicated by STAT symbol "R") then almost immediately begins to idle (indicated by STAT symbol "I"). Eventually, all of the processes on the backend controller are "killed" (Exit Processor), indicating that the system has crashed and the transaction can not be executed. The two transactions were tested in all three of the test transaction files and similar problems were encountered.

In analyzing the cause of this problem, our initial theory was that in the hashing module used by the retrieve-common operation, the following problems may exist:

* The hashing operation may not be functioning properly due to a faulty addressing scheme created to store the retrieved records from the source and target retrieve.
* There could be record collisions as a result of hashing to which have not been addressed by the hashing algorithm.
* The hashing operation may not be properly handling bucket overflow.

After tracing and debugging the code, it was discovered that there was a memory allocation problem stemming from the hashing and bucket_block procedures of the hashing module.

a. The Hashing Procedure

This procedure is used to perform the hashing operation on the values of the join attributes of the input records which are either the backends' local source records or the local target records. The output from the procedure are the input records and their hashed values (i.e., the bucket numbers), which are sent to the bucket-block procedure with the request **id** for further processing [Ref. 8]. The hashing operation is done by the hashing functions of this procedure.

A hashing table with a large number of buckets is useful for a number of reasons. First, the large number of buckets may reduce the chance of hashing different attribute values of records into the same buckets. Second, the number of records in each bucket is also quite small and minimizes the overflow issues. Thus it will reduce the access time during merging. The **bucket index** of a hashing table is an array of fixed-size bucket entries. There is a bucket entry for each bucket to keep track of the records which are stored in that bucket [Ref. 8].

In the case of MBDS, 16K bytes were determined to be the size of the hashing table, yielding 2048 entries (therefore, 2048 buckets) in the hashing table each with a bucket entry size of 8 bytes. Each of the retrieve-common requests requires two hashing tables, one for the source retrieve records and one for the target retrieve records.

Because of the potentially large number of hashing tables concurrently in use, it is necessary to store the bucket indexes of the tables in the secondary storage and stage them into the primary memory on demand [Ref. 8].

b. Problems with the Bucket Number Constant

The memory allocation problem appears to be centered around the **number of buckets** specified in the Hash_Function procedure. The value of the "Bucket Number" constant was set to 2048 buckets as determined in [Ref. 8]. This constant is defined in /@db4/u/mdbs/6/common/commdata.def as the maximum number of buckets for the hash table.

When the test transactions are ran on a two backend configuration, the system crashes, failing to process the transactions. The trace files on the program execution reveals that there is a problem with memory allocation. The error message reads; "Problem with Malloc in bucket block." This error message is a result of the following code;

```
if ((blk=(struct block *)
malloc (size of struct block)))==Null)
print if ("problem with malloc in bucket block")
```

Another error message reads as follows:

SYSTEM ERROR 1: Problem with Malloc in create_BE Target Info-node ().

Looking at the trace files produced, it was obvious that the limit set on the number of buckets was exceeded by the number of records being retrieved from the two test transactions.

The second retieve-common test transaction in the medium data record transaction file (SDB-MDR#1) accesses all of the records from the medium record set (8400 records) and the small record set (16,800 records) such that one-half (1/2) the records satisfy each retrieve (source & target). In this case there is a total of 12,600 records being retrieved into the buckets of the hash table. Clearly this number of records drastically exceeds the 2048 buckets created in the hash table for storage of the source and target records. The third test transaction also exceeds the limit on the number of buckets as it retrieves a total of 5250 records for temporary storage.

The second and third retrieve-common test transactions in the medium-large record transaction file (SDB_MLR#1) retrieve totals of 5880 and 2100 records, respectively, which have also exceeded the limit set on the number of buckets for storage.

The test transactions in the large record transaction file (SDB_LGR#1) were able to run but not consistently, due to memory not being freed as one transaction is completed. This problem is further discussed in the next section.

c. Other Problems Encountered with Retrieve-Common

In an attempt to correct the problems with the bucket number constant, other severe problems were discovered. There were two proposed solutions to the bucket number problem:

* increase the bucket number constants to whatever is needed, and

* keep the current constants and reduce the number of records retrieved by the two retrieve-common transactions in each test transaction file.

In attempting the first solution, the bucket number constant had to be increased by powers of two. This is a standard practice for the UNIX system where the bucket number of the hash tables are established using base two (2) (i.e. 2048 is 2^{11}). The bucket number was increased to 8192 buckets (2^{13}) and then 16384 buckets (2^{14}).

This tremendous increase in buckets created a larger bucket index and subsequently a much larger hashing table to be maintained by the system. Although the system uses the virtual memory which is virtually unlimited, problems were still encountered. The system seemed to be overloaded by the vast amount of paging of buckets in and out of primary memory to accommodate the storage of the retrieved records. This swapping is inevitable because the system does not have enough primary memory to handle the amount of buckets required to store the records retrieved by the retrieve-common operations.

Subsequently, the second solution was attempted which resulted in the discovery of more problems with the retrieve-common. The two transactions of each test transaction file were reduced down to 25% of the records being accessed. This reduction worked for the first run, but

subsequent runs were not successful. This problem resulted from the allocated memory not being freed after a transaction has been processed. There was also another case, where the third transaction would run partially, but would hang up, outputting only a portion of the query results. This problem is caused by a missing message which indicates a problem in communications between the controller and the backends when conducting the final phase of the retrieve-common operation.

7. Proposed Solutions to Existing Problems

In addressing the memory allocation problems of the original transactions produced by CABS, two solutions are proposed:

* More primary memory should be added to the system in order to process the retrieve-common transactions involving the retrieval of large amounts of records. A substantial increase in random access memory (RAM) will reduce the requirement for massive paging as it provides the required temporary storage for the retrieval of source and target records.

* Another solution to this problem would be a complete algorithmic approach to the hashing operations (i.e., map different attribute values of records into the same bucket addresses), in lieu of the hashing table. The correspondence between the hashed values and bucket addresses is no longer represented in a hash table which requires considerable space. By employing the algorithm, we can dynamically compute the correspondence without the need of any space for the table.

Attempt was made by the system programmers to correct the problems with the freed memory, and loss of messages. However, due to the lack of time these problems were not

solved in order to allow proper testing of the retrieve-common operation.

Future research on the retrieve-common operation must involve a complete evaluation and thorough clean-up of the existing hashing algorithm to minimize the use of either virtual and primary memories.

B. THE UPDATE OPERATION

1. An Overview of Update

An update request is used to modify the attribute values of records in the database. The update request consists of two parts: a query and a modifier. The query identifies which records of the database are to be modified and the modifier specifies how the records are to be modified. The following is an example of an update request:

UPDATE((TEMP=Cors) and (CNUM=C200))<Room=S429>

This update request would change the room for course number (C200) in the course (cors) file to S429.

2. The Design and Execution of an Update Request

The following design of the Update transaction is described in [Ref. 10]. The modifier which specifies the new value to be taken by the attribute being modified may be one of the types described below:

```
Type - 0   : <attribute=constant>
Type - I   : <attribute=f(attribute)>
Type - II  : <attribute=f(attribute-1)>
Type - III : <attribute=f(attribute-1) of Query>
Type - IV  : <attribute=f(attribute-1) of Pointer>.
```

In the simplest case, a modifier indicates the new value to be taken by the attribute being modified (i.e., type-0). In the more involved cases, the modifiers specify the new value to be taken by the attribute being modified as a function f of the 'old' value of that attribute (i.e., type-I) or values of some other attribute of the record to be updated (e,g., types II, III or IV). The other attribute is called the **base attribute** (i.e., attribute-1 in the specification).

In this study, we will only evaluate an update request containing modifiers of type-0. The program code for update requests containing modifiers of type III, and IV was not available for this study.

An update request containing a modifier of type-0 is broadcasted by the controller to all the backends. The backends will perform descriptor processing and address generation. Afterwards, each backend has a list of secondary memory addresses of the tracks containing the relevant records. These tracks are accessed by respective backends and the records satisfying the query are selected from these tracks. These are the records to be updated.

Each of these records is updated using the modifier in the update request. If the modifier is of type-0, the new value to be taken by the attribute being modified in a record to be updated is provided in the modifier.

Based on the attribute value that is changed, an updated record may remain in the same cluster which it

belonged (its pre-updated version) or it may now belong to a different cluster. In the latter case, a record is said to **change cluster**. An updated record will belong to a different cluster only if the set of descriptors from which it is derived is different from the set of descriptors from which the pre-updated version was derived. If the modifier is not a directory attribute, the updated record continues to be derived from the same set of descriptors, since only directory attributes affect the descriptors. Hence, the update record does not change cluster. Such update is termed **simple update**. If the modifier is a directory attribute, an updated record may change cluster. This type of update is called a **complex update**. In this study we will only evaluate the simple update since the program code for complex update was not available.

In order to check whether or not a newly updated record changes cluster, it is necessary for a backend to search the descriptor-to-descriptor-id table (DDIT). To facilitate such search, we have decided that each backend should replicate the descriptors for all the directory attributes in its secondary memory.

Finally, each backend will send an acknowledgement to the controller to indicate that it has finished processing the update request. When it has received acknowledgements from all backends, the controller will output a message to the user to signal successful completion of the update request.

3. The MBDS Test Transactions

It was discovered in this study, that CABS did not generate any test transactions for updates in the transaction set files. Thus, the update transactions had to be manually constructed in four separate files for each of the record classes. These files are named: lgr#2, mlr#2, mdr#2, and smr#2 for their respective record class.

The test transactions were constructed according to the request set proposed in [Ref. 4]. Each test transaction file is comprised of three update requests, all which are data-intensive. The first update request identifies 1/8 of the database (record class) and updates STR0001's value from "Xxxxxxxxx" to "Oneeighth". The second update request identifies 1/4 of the database (record class) and updates STR0005's value from "Xxxxxxxxx" to "Onequartr". The third update request identifies 1/2 of the database (record class) and updates STR0010's value from "Xxxxxxxxx" to "Onehalfff".

4. Existing Problems with the Update Request

While testing the update transactions described above, a problem with the memory was encountered, similar to the problem in the retrieve-common operation.

When trying to run the three transactions in the respective test transaction files, the system attempts to run the transactions, then it suddenly begins to idle and subsequently kills the processes on the controller and the backends.

A trace on the program execution revealed the following error message, "not enough memory." The problem was determined to be involving a constant defined in the program code, named "MAX-ADDRS-UPD". This constant establishes the list of secondary memory addresses for temporary storage of the records to be updated. This constant was originally set to one (1) which was established to update only one record at a time.

In an attempt to solve this problem, this constant was set to 100. Another constant, named "MAX_DIO_REG", was also modified based on the "MAX_ADDRS_UPD". The MAX_DIO_REG constant was redefined by multiplying MAX_ADDRS_UPD by a certain value (i.e., MAX_ADDRS_UPD*4). When MAX_DIO_REG was defined as MAX_ADDRS_UPD*4, the first test transaction of the large record file (lgr#1) ran successfully. This transaction is designed to update only 210 records. The remainder of the test transactions in this file and the other files were not able to run with the constants defined as such.

Therefore, MAX_ADDRS_UPD was multiplied by eight (8), six (6), and five (5), respectively. These constant modifications still did not allow the remainder of the test transactions to run successfully.

The final assessment of the problem, is that the number of records being retrieved by the update transaction requires a fairly large amount of storage space on the single backend configuration. Since there is not enough primary

memory (RAM) to store all the records at one time, the system is once again overloaded by an intense amount of paging of records in and out of the buffer. This paging subsequently causes the system to crash, killing all of the controller and backend processes. There once again, seems to be a problem with the system freeing memory of completed transactions. Transactions updating 420 records run inconsistently due to memory not being freed, as with the retrieve-common.

5. Proposed Solutions to the Problems

In proposing solutions to the problems and with the update operation, we must again consider adding more primary memory to the system in order to reduce the intense requirement for paging of records in and out of memory. The problem with the system not freeing memory must also be closely observed, and this portion of the code must be fixed.

Future implementors of the MBDS update may also consider preparation of each update request to update the desired amount of records in sessions in contrast with the entire amount of records at one time. In this way, the buffer size for the updating records can be small and fixed, although the number of records required updating may be large and variable.

V. THE PRELIMINARY RESULTS

A. A SUMMARY OF RESULTS

The preliminary results provided in this section are to verify the successful modification to a portion of the program code for the update operation. Initially, the simple update would work only when trying to run transactions that would update a single record. As discussed in Chapter IV, changes were made to two of the constants used to establish the list of secondary memory addresses for temporary storage of the records to be updated. This modification allowed the first test transaction of the large-record file (lgr#1) to be executed, updating only a small amount of records (210 records). This number of records was also used to test the other three record files.

The results obtained from the preliminary testing of this single transaction is provided in this chapter as the preliminary results. Although these results verify some success in the modification of the update code, they are not conclusive enough to verify any positive performance improvements in regards to response-time reduction (RTR). The response times also tend to level off after a two-backend configuration was tested. This is probably caused only by this small amount of records being distributed over two of the backends. Any increase in response time is caused by the

overhead intensity involved in using more than the required number of backends for this small number of records. Due to the memory problems encountered during the preliminary testing, the response-time invariance (RTI) claim which requires the database size to be multiplied, was not considered for preliminary testing. Only transactions for response-time reductions were tested throughout this study.

The preliminary testing for the simple update operation was only conducted on as many backends as in a four-backend configuration. Only a total of six backends were available for testing because of disk problems with two of the eight backends. Due to the lack of time, preliminary testing was conducted on only four (4) of the six (6) available backends. The results are presented in the following sections.

1. **The Single-Backend Configuration**

The following preliminary times recorded for the response-time reduction of a single transaction are based on the single-backend configuration. Table 1 lists the transaction response times in seconds, while Table 2 lists the number of records of each record size.

TABLE 1. THE SINGLE-BACKEND RESPONSE TIMES

Trans #	SMR	MDR	MLR	LGR
TR 1	4.350	5.400	9.200	14.983

TABLE 2. THE SINGLE-BACKEND RECORD DISTRIBUTIONS

Rec Size	BE #1
Large	1680
Med-Large	3360
Medium	8400
Small	16800
Total	30240

2. The Two-Backend Configuration

The following preliminary times recorded for the response-time reduction of a single transaction are based on the two-backend configuration. Table 3 lists the transaction response times in seconds, while Table 4 lists the number of records of each record size.

TABLE 3. THE TWO-BACKEND RTR TEST

Trans #	SMR	MDR	MLR	LGR
TR 1	4.366	3.483	5.233	9.100

TABLE 4. TWO-BACKEND RECORD DISTRIBUTION

Rec Size	BE #1	BE #2	Total
Large	810	870	1680
Med-Large	1648	1712	3360
Medium	4198	4202	8400
Small	9050	7750	16800
Total	15706	14534	30240

3. The Three-Backend Configuration

The following preliminary times recorded for the response-time reduction of a single transaction are based on the three-backend configuration. Table 5 lists the transaction response times in seconds, while Table 6 lists the number of records of each record size.

TABLE 5. THE THREE-BACKEND RTR TEST

Trans #	SMR	MDR	MLR	LGR
TR 1	5.283	6.400	5.416	8.633

TABLE 6. THREE-BACKEND RECORD DISTRIBUTION

Rec Size	BE #1	BE #2	BE #3	Total
Large	528	582	570	1680
Med-Large	1168	1008	1184	3360
Medium	2772	2648	2980	8400
Small	4663	5476	6661	16800
Total	9131	9714	11395	30240

4. The Four-Backend Configuration

The following preliminary times recorded for the response-time reduction of a single transaction are based on the four-backend configuration. Table 7 lists the transaction response times in seconds, while Table 8 lists the number of records of each record size.

TABLE 7. THE FOUR-BACKEND RTR TEST

Trans #	SMR	MDR	MLR	LGR
TR 1	7.200	5.233	3.850	6.183

TABLE 8. FOUR-BACKEND RECORD DISTRIBUTION

Rec Size	BE #1	BE #2	BE #3	BE #4	Total
Large	440	364	370	506	1680
Med-Large	876	820	772	892	3360
Medium	1907	2144	2291	2058	8400
Small	4117	3818	4933	3932	16800
Total	7340	7146	8366	7388	30240

B. A COMPARATIVE CHART

Figure 4 shows the RTR performance of MBDS on the update transaction. The response times are taken from the update transaction in the large record transaction file. This chart reveals some positive indications of performance improvements over the two and four backend configurations.

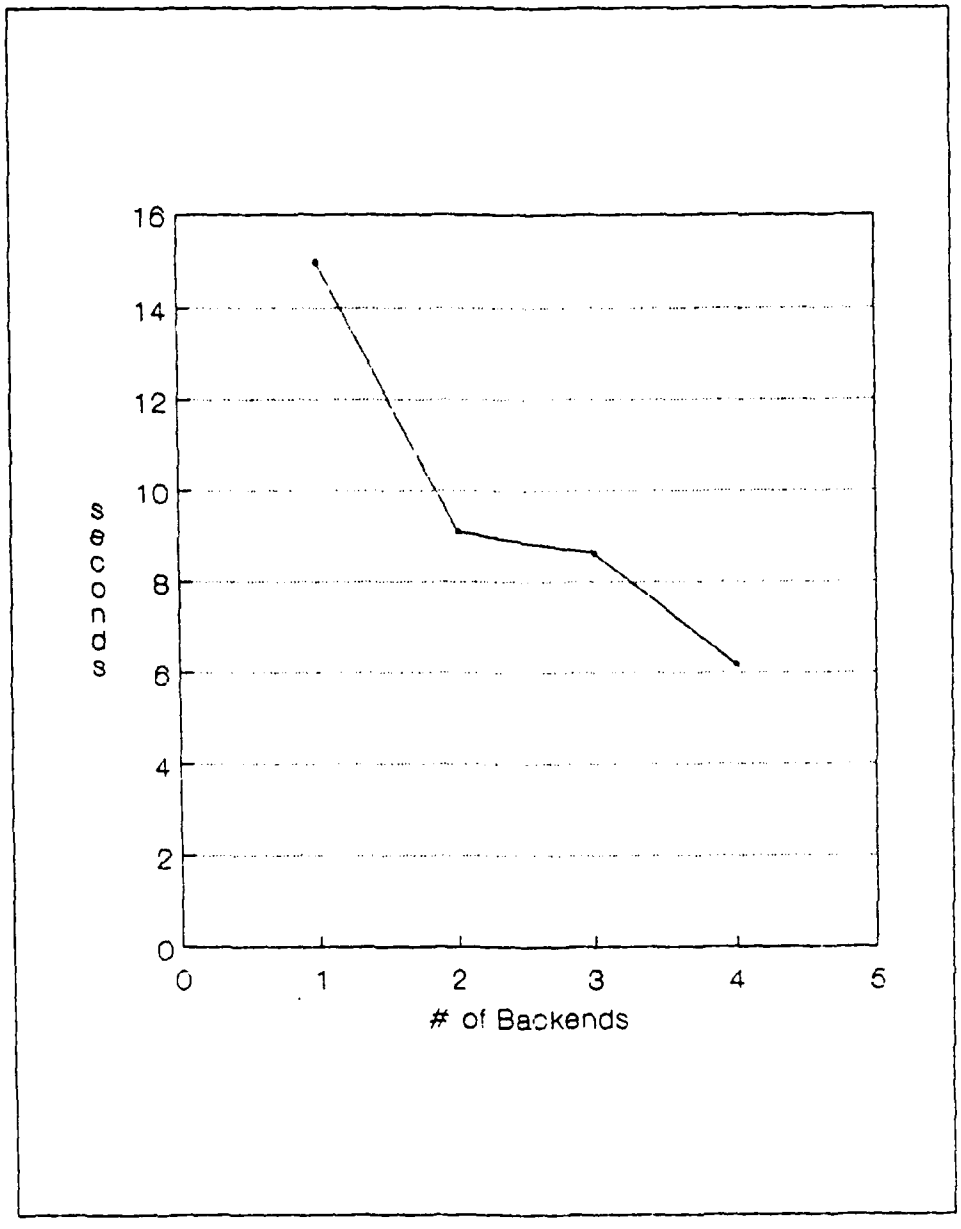


Figure 4 RTR Performance on Transaction #1

VI. CONCLUSIONS AND RECOMMENDATIONS

The main emphasis of this thesis was on the instrumentation of a parallel and scalable database computer, namely, the Multi-Backend Database System (MBDS), for benchmarking its complex operations. The primary research question was to determine whether MBDS demonstrates the response-time reduction and response-time invariance when carrying out its two complex operations: UPDATE and RETRIEVE-COMMON. In order to benchmark these transactions, the proper instrumentation of the test database, test transaction sets, and test procedures was thoroughly executed. Detailed testing, problem identification (debugging), and minor software modifications were also conducted in an attempt to verify the correctness of the program code for these complex operations.

During the preliminary testing of the UPDATE and RETRIEVE-COMMON operations, several major problems were discovered in the program code. Ironically, the problems encountered with both operations were centered around memory allocations. The program code for each operation includes a constant which is used to set and define a desired data structure (i.e., bucket index, and address array) of a particular size for adequate storage of large amounts of records being retrieved by both

operations. The data structures provide a buffer for selected records to be stored.

Although the system uses the virtual memory, there is not enough main memory available to minimize the amount of paging of data structures and their retrieved records, in and out of the main memory. Of the four (4) megabytes of the main memory provided per backend, the UNIX system takes up approximately 1.5 megabytes alone, leaving only 2.5 megabytes for storage of large amounts of records from 100 to 1000 bytes in size, with their associated data structures which include pointers, link lists, address arrays, bucket indexes, block structures, etc. Thus, it appears that the key to successful benchmarking of the UPDATE and RETRIEVE-COMMON operations rest on the proposal to substantially increase the amount of the main memory per backend which is presently four (4) megabytes. A second alternative is to consider a lengthy algorithmic change to the storage algorithms used by each operation.

Nevertheless, MBDS has made tremendous strides in meeting its designed performance goals in regards to the RETRIEVE and DELETE operations. However, some short-term efforts will be required from highly qualified system programmers to examine and clean up the existing software problems encountered while testing the UPDATE and RETRIEVE-COMMON operations. Once the evaluation, modification, and testing of the software has been successfully completed, and correctness and proper functioning of the program code has been verified, formal benchmarking of

these complex operations can then be thoroughly exercised. Subsequently, the results of the benchmarks can be used to fully assess the performance claims of the implementor of MBDS based on all four of its operations: RETRIEVE, DELETE, UPDATE, and RETRIEVE-COMMON.

LIST OF REFERENCES

1. Hsiao, David K. and Kamel, M. N., "Heterogeneous Databases: Proliferations, Issues and Solutions," IEEE Transactions on Knowledge and Data Engineering, KDE-1,1, 1989.

2. Vincent, J. R., A Performance Measurement Methodology for Software Multiple-Backend Database Systems, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1985.

3. Hall, James E., Performance Evaluation of a Parallel and Expandable Database Computer--The Multi-Backend Database Computer, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1989.

4. Fenton, G. P., A Computer Aided Design for the Generation of Test Transactions and Test Databases and for the Benchmarking of Parallel, Multiple-Backend Database Systems, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1986.

5. Demurjian, S. A., Hsiao, D. K., and Menon, M. J., "A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Upgrade," Proceedings of the Second International Conference on Data Engineering, IEEE Computer Society Press, February 1986.

6. Strawser, P. R., A Methodology for Benchmarking Relational Database Machines, Ph.D. Dissertation, The Ohio State University, Columbus, OH, 1984.

7. Tekampe, R. C. and Watson R. J., Internal and External Performance Measurement Methodologies for Database Systems, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1984.

8. Tung, H., Design, Analysis and Implementation of the Primary Operation Retrieve-Common of the Multi-Backend Database System (MBDS), Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1985.

9. Hunt, A.L., Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS), Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1986.

10. Hsiao, D. K., and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Advanced Database Machine Architecture, pp. 327-385, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department of The Army U.S. Total Army Personnel Command TAPC-OPF-SC ATTN: Chief, Signal Branch 200 Stovall Street Alexandria, VA 22332-0415	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5002	1
5.	Curriculum Officer, Code 37 Computer Technology Programs Naval Postgraduate School Monterey, CA 93943-5002	1
6.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	5
7.	Captain Darrell W. Alston Headquarters, TRADOC Deputy Chief of Staff, Resource Management Division Support Systems Ft. Monroe, VA 23651-5000	1
8.	Marciano Pitargue Vitalink Communications Corporation 6607 Kaiser Drive Fremont, CA 94555	1
9.	Thomas Chu 1140 Pebblewood Way San Mateo, CA 94403	1